Python Introduction to Plotting

Main Points

1 Background

Plots are very helpful for presenting the results of many engineering calculations. Plotting with *Python* and the *Matplotlib* library is fairly easy to do. The *Matplotlib* (https://matplotlib.org) library can create many types of publication quality plots. *Matplotlib's* `pyplot` module was created to make *MATLAB* type plots in *Python*.

- Plotting commands can be used from...
  - REPL (command line)
  - Scripts and functions
  - *Jupyter* notebooks
- Sequences of numeric values are required to create plots
  - Lists
  - Tuples
  - Ranges
  - *NumPy* arrays
- Sequence data can be collected from an external source or created using math
- The simplest plot can be created using two lists (`x` and `y`) and issuing just a few commands
- *Matplotlib* provides many possible ways to modify and add to the plot, they include...
  - Setting axes limits
  - Adding a title and axes labels
  - Turning on a grid
  - Setting line types, colors and widths
  - Setting marker shapes and sizes
  - Adding legends
- Axes may be linear or log10 or a combination of the two

- Specialty plots can be created, such as...
    - Polar plots
    - Histograms
    - Bar charts
    - Pie charts
    - Stairs plots
    - Stem plots
- Plots may be combined into the same plot window or multiple windows can be used

Approaches for Using *Matplotlib* and the `pyplot` module

1. State-machine approach based on the commands used in *MATLAB* to create plots
2. Object-oriented approach
    - Considered the more *Pythonic* approach because it is more explicit
    - The approach used in this document
    - Generally requires a bit more coding, but not a lot
    - Some results can only be achieved by using this approach
    - For many plotting tasks either approach will work

2 Creating Lists of Values for Plotting

The easiest way to create values for plots is to use *NumPy* array (presented at another time). However, *Python* can be used to do the same thing with just a little extra work. This section describes how to prepare lists of values for plotting by creating a pair of special-built functions and list comprehensions.

2.1 Ranges with Non-integer Values

The `range()` function only works with integer values for the starting, ending, and step size values and will only contain integers. When plotting it is often desirable to have a list with a non-integer values. The following sub-sections demonstrate the creation of two functions that can be used instead of the standard `range()` function. One creates a list with a specified number of values and the other a list with a specified non-integer step size.

2.1.1 Specified Number of Values - `frange()`

Presented below is a mathematical expression for creating a list of equally spaced values between upper and lower limits.

$$x_i = (x_{upper} - x_{lower})\frac{i}{(n-1)} + x_{lower}$$

Where...

- i is the index position of a zero indexed list
- x_{lower} and x_{upper} are the desired lower and upper limits
- n is the desired number of values in the list

A function named `frange(lower, upper, n=100)` (short for floating point range) that uses the above expression has been created in the following code block. The function uses a list comprehension to to create the list that the function returns.

- Returns a list containing `n` floating point values
- Between limits of `lower` and `upper` (including both limits)
- Last argument `n=100` is a keyword argument
    - Argument is optional
    - Default `n` will be 100 if argument is not used
    - Include `n=` or not; i.e. `n=100` and `100` both work

Creating and testing the `frange()` function

```
def frange(lower, upper, n=100):
    return [(upper - lower) * i/(n - 1) + lower for i in range(n)]

a = frange(-2, 3, 11)
print(a)
print(f"Length of list 'a' is {len(a)}")
```

```
[-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
Length of list 'a' is 11
```

2.1.2 Specified Step Size – `step_range()`

The `step_range(start, stop, step)` function has been created to return a "range" of values with a non-integer step size. The function actually returns a list, not a true range. Lists created using `step_range()` will end with the nearest full step at or before the `stop` value. This means that `stop` value will only be included in the list if `step` divides evenly into `stop - step`. The function rounds the values in the list to 8-decimal places.

The `step_range()` function

```
def step_range(start, stop, step):
    n = int((stop - start) / step + 1)
    stop = (n - 1) * step + start
    x = [round((stop - start) * i/(n - 1) + start, 8) for i in range(n)]
    return x

x = step_range(0, 20, 2.5)
print(x)
```

```
[0.0, 2.5, 5.0, 7.5, 10.0, 12.5, 15.0, 17.5, 20.0]
```

2.2 Defining Functions and Using Them in List Comprehensions

Sequences of both x-values and y-values are usually needed for creating plots. Many times the x-values are used to calculate the y-values.

- Sequences of x-values could be created...
  - Manually as a list, tuple, or array
  - With a standard `range()` command for integer values, i.e. `x = range(10)`
  - With a custom function like `frange()` or `step_range()`
  - With *NumPy* (covered separately)
- Use the sequence of x-values to create y-values
  - Standard *Python*
    - Define a function that returns a calculated value
    - Use a list comprehension that calls the function with the x-values
    - See the example code block below
  - Use *NumPy* array math

An Example

The following code block uses the previously created list named `a` and a list comprehension to create a list named `b` from the following expression:

$$b = 3.5^{(-0.5a)} \cos(6a)$$

A example using a function and list comprehension

```
import math
def bfunc(a):
    return 3.5*(-0.5*a) * math.cos(6*a)

b = [bfunc(a) for a in a]
print(f"a = {a}")
print(f"b = {b}")
```

```
a = [-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
b = [2.9534888555637226, -2.391716937447277, 1.6802980016381404,
↪  -0.8662434345253898, -0.0, 0.8662434345253898, -1.6802980016381404,
↪  2.391716937447277, -2.9534888555637226, 3.3236346187573433,
↪  -3.466662718281421]
```

3 Getting Started with *Matplotlib*

The ability to create plots is not built into the *Python* standard library. Plotting functionality is provided separately by the excellent *Matplotlib* library (among others). This document uses *Matplotlib's* `pyplot` module for all plotting operations.

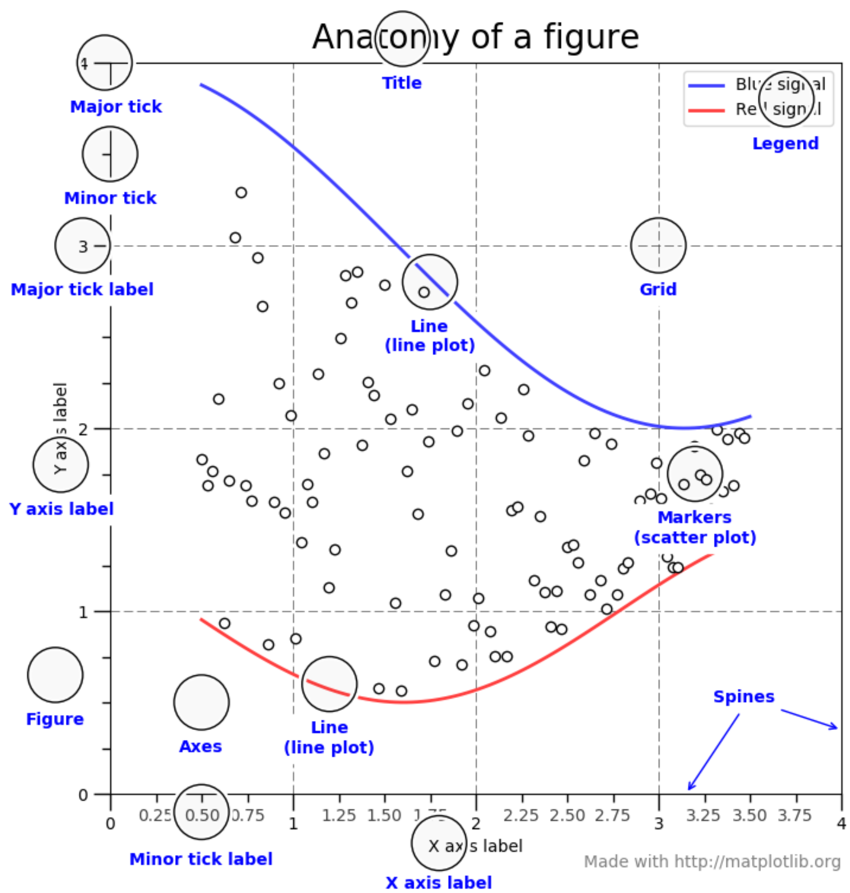
- `pyplot` must be imported before issuing any plotting commands; two common methods are...
  - `import matplotlib.pyplot as plt`
  - `from matplotlib import pyplot as plt`
- There are two fundamental approaches for creating plots using *Matplotlib*
  - Object-oriented: more *Pythonic* and explicit
  - State-machine: more *MATLAB*-like and less explicit
- This document will exclusively use the object-oriented approach

Typical imports for plot creation

```
# usually either math or numpy, not both
import math
import numpy as np
import matplotlib.pyplot as plt
```

Matplotlib uses specific names for the parts of a plot. The figure below comes from the *Matplotlib* website and illustrates the many different parts of a plot figure.

- The plot itself is referred to as an *Axes*
- Axes are located within a *Figure*
- Object-oriented approach gives fine control over both the figure and the axes added to it



4 Plotting Basics

One of the simplest possible plots is an x, y plot using sequences (i.e. lists) of x and y values. A figure and axes must be created before plotting when using the object-oriented approach. The following three methods could be used to plot x, y pairs with essentially the same results.

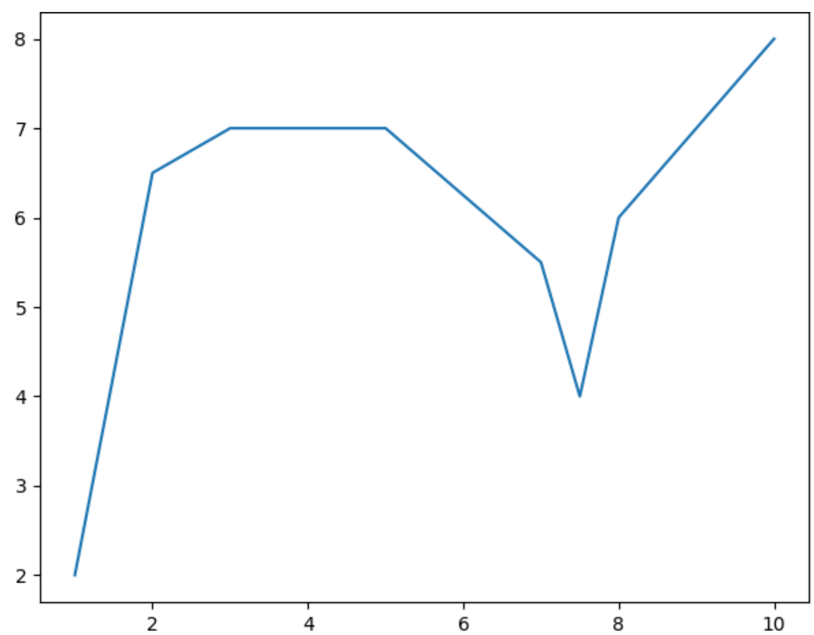
4.1 Method 1

- Create a blank figure (often named `fig`) using `plt.figure()`
- Add an axes object (often named `ax`) to the figure using `fig.add_axes()`
- Use list of location/size values as argument in `fig.add_axes()`
  - `[x-start, y-start, x-end, y-end]`
  - Floats between 0 and 1 for each item
  - Values are relative to the lower, left corner of figure
  - $x, y = (0, 0)$ is the lower, left corner
  - $x, y = (1, 1)$ is the upper, right corner
  - Example: `[0, 0, 1, 1]` fills the space from lower, left to upper, right of the figure
- Plot the values to the axes object using `ax.plot(x, y)`

Method 1 – using `plt.figure()` and `.add_axes()`

```
x = [1, 2, 3, 5, 7, 7.5, 8, 10]
y = [2, 6.5, 7, 7, 5.5, 4, 6, 8]

fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
ax.plot(x, y)
```

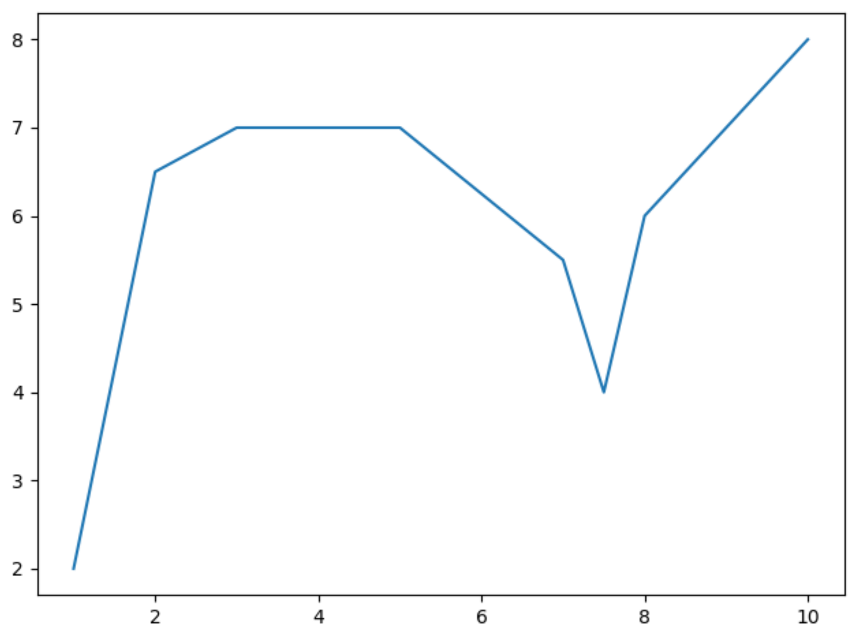


4.2 Method 2

- Create a blank figure (often named `fig`)
- Add a subplot object (often named `ax`) using `fig.add_subplots()`
- Subplot objects allow multiple axes (plots) to be arranged in different ways...
  - Next to each other
  - Above/below each other
  - In a grid
- `fig.add_suplots()` accepts a 3-digit argument to set the axes arrangement
  - First digit; number of axes high (rows)
  - Second digit; number of axes wide (columns)
  - Third digit; the current axes for plotting
- Example, `ax = fig.add_subplot(121)` means...
  - 1 plot (axes) high (1 row)
  - 2 plots (axes) wide (2 columns)
  - The current plot (axes) is 1 (the top)
  - Subplot group assigned to the name `ax`
- Plot the values to the axes with `ax.plot(x, y)`

Method 2 – using `plt.figure()` and `.add_subplot()`

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
```

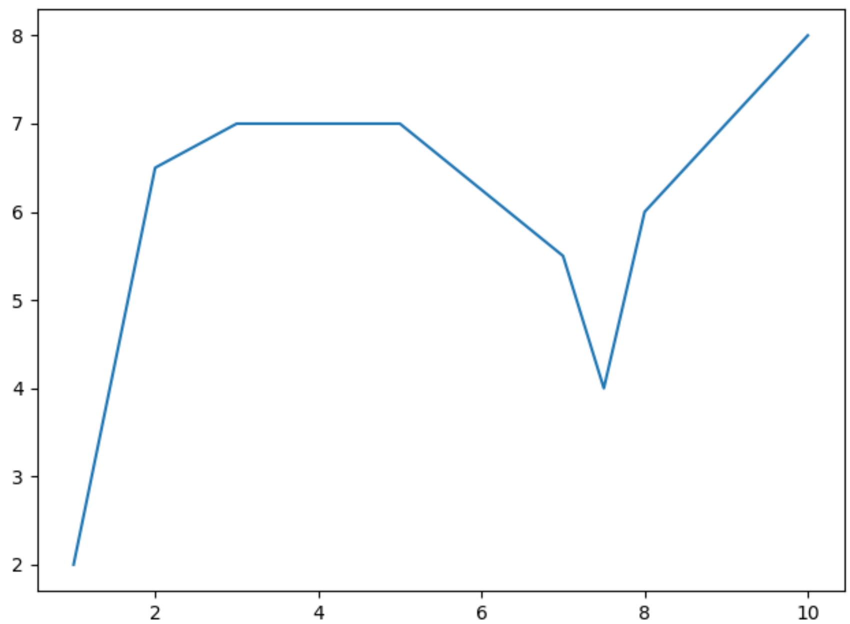


4.3 Method 3 – the preferred method

- Create a figure and axes at the same time by assigning `plt.subplots()` to a tuple with the figure and axes names
- Only be single axes will be created if no arguments are included in `plt.subplots()`
- May include a number of named arguments in `plt.subplots()` – later
- Plot the values to the axes `ax`

Method 3 – using `plt.subplots()`

```
fig, ax = plt.subplots()
ax.plot(x, y)
```



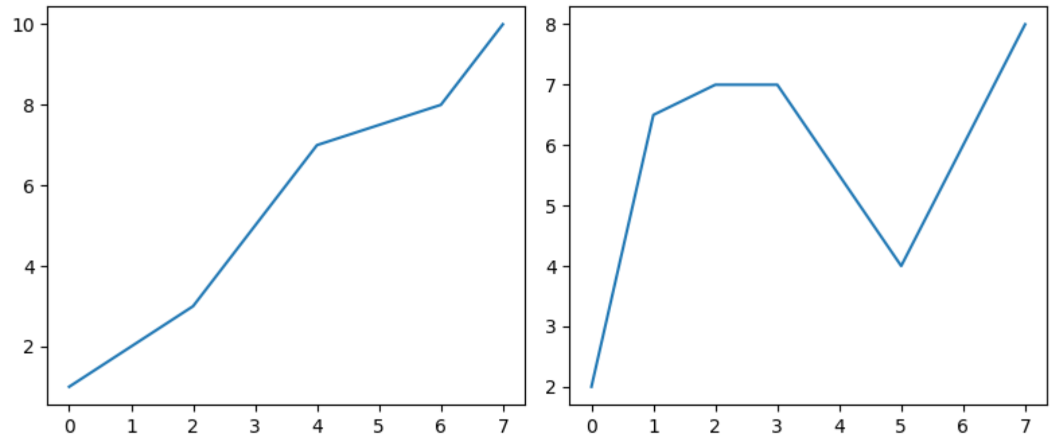
4.4 More Plotting Basics Information

- Variable names used in the `plot()` function do not need to be `x` and `y`
  - Both must be sequences of the same size
  - First variable in the `plot` will be used to generate the x-values of each plotted point
  - Second variable will be used to generate the y-values of each plotted point
- It is possible to generate a plot with a single list; `plot(x)` or `plot(y)`
  - The list name used in `plot()` will correspond to the y-values of the plotted points
  - The x-value of the plotted points will match their index positions in the list
  - Plotting a single variable is not common for engineering problems

The following code block demonstrates what happens when only `x` or `y` is used in the plotting command. The example is using a subplot that is 1 row high by 2 columns wide in order to place the plots next to each other.

Plotting a single variable using a 2-column subplot

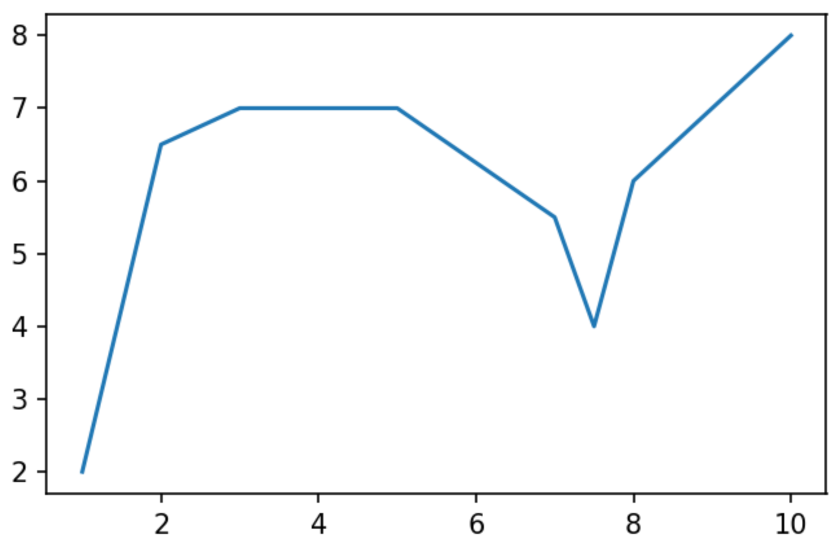
```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot(x)
ax2.plot(y)
```



- The size of a figure can be set using keyword arguments in `plt.subplots()`
  - `figsize=(x, y)` where `x, y` is the size in inches
  - `dpi=xxx` where `xxx` is the pixel density in dots per inch

Setting `figsize` and `dpi` when using `plt.subplots()`

```
fig, ax = plt.subplots(figsize=(4.5, 3), dpi=150)
ax.plot(x, y)
```



5 Formatting Arguments for `.plot()`

The `.plot()` method accepts a number of optional arguments that modify the look of a plot. The *Matplotlib* documentation includes a comprehensive list of the options.

- Quick line style, color, and marker changes
  - Place a formatting string immediately after the sequence names
    - Include values for any or all the options (style, color, and marker) in any order
    - The string `'-r'` will create a solid red line with no markers
    - The string `'--bo'` will create a dashed blue line with circle markers at each data point
  - Valid line types are...
    - Solid is `-`
    - Dashed is `--`
    - Dash-dot is `-.`
    - Dotted is `:`
  - Valid colors include...
    - Red is `r`
    - Green is `g`
    - Blue is `b`
    - Cyan is `c`
    - Magenta is `m`
    - Yellow is `y`
    - Black is `k`
    - White is `w`
  - Valid marker shapes include any of the following, plus more
    - Point is `.`
    - $\times$ symbols are `x` and `X`
    - Plus signs are `+` and `P`
    - Circle is `o` = circle
    - Diamonds are `d` and `D`
    - Square is `s`
    - Triangles are `^`, `v`, `<`, and `>`
    - Hexagons are `h` and `H`
    - Star is `*`
    - Pentagon is `p`
    - Octagon is `8`
- Keyword arguments can be added after the formatting string (if used)
  - `linewidth=2.0` or `lw=2.0` will make the plotted line 2-points wide
  - Line style can be changed using `linestyle='--'` or `ls='--'`
  - Use `marker='*'` to set the marker type
  - Use `markersize=3` or `ms=3` to set the marker size to 3
  - Colors can be changed using `color='r'` or `c='r'`
  - Using `color` or `c` allows for a much broader range of colors by typing color names as strings
- Use `plt.show()` after creating plots to keep from also showing a plot object descriptor

The code block below plots 100 values of (a, b) for a from -2 to 4 where b is calculated using the following expression...

$$b = 3.5^{(-0.5a)} \cos(6a)$$

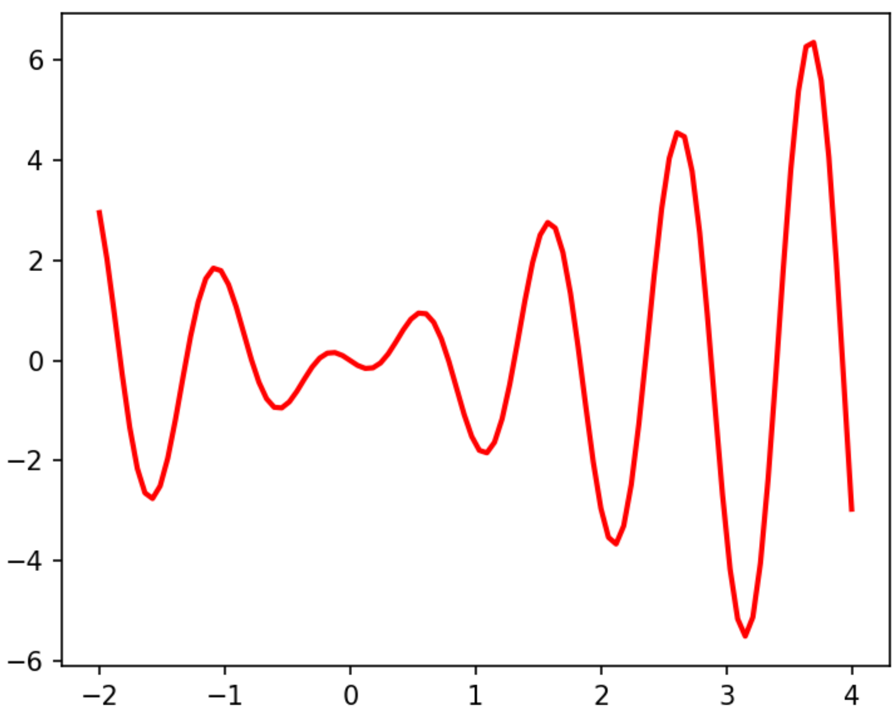
The plot is formatted with a solid red line that is 2.0 wide.

Plot with line color, style, and width settings

```
def frange(lower, upper, n=100):
    return [(upper - lower) * i/(n - 1) + lower for i in range(n)]

def bfunc(a):
    return 3.5*(-0.5*a) * math.cos(6*a)

a = frange(-2, 4)
b = [bfunc(a) for a in a]
fig, ax = plt.subplots(figsize=(5, 4), dpi=150)
ax.plot(a, b, '-r', lw=2.0)
plt.show()
```



6 Adding a Title, Labels, and Other Stuff

It is quite easy to add a title, axes labels, and other options to plots. The Matplotlib Axes class lists all of the possible options. All code snippets below assume a figure named `fig` has a subplot object named `ax`. All formatting commands should be placed after the `.plot()` function and before `plt.show()`.

- Add titles and axis labels
  - Add a title using `ax.set_title("Title Text")`
  - Add axis labels with `ax.set_xlabel("x")` and `ax.set_ylabel("y")`
  - Set label/title text height in points by including the `fontsize=x` keyword argument
  - Use `ax.set(title='Title', xlabel='x label', ylabel='y label')` to set all three label strings at one time
- Set axis limits
  - Using `ax.axis([xmin, xmax, ymin, ymax])` sets the plot limits in the x and y directions (requires that all 4 values be given, i.e. `ax.axis([-2, 4, -7, 7])`)
  - Use named arguments like `ax.axis(xmin=0, xmax=10, ymin=-4, ymax=5)` instead to change any or all axes limits
  - Use `ax.set_xlim(xmin, xmax)` or `ax.set_ylim(ymin, ymax)` to set limits for just one axis
- Add a grid by using `ax.grid(True)` or just `ax.grid()`

The example code block below uses *NumPy* instead of list comprehensions to create the sequences `x` and `y` for plotting the following expression with the listed options.

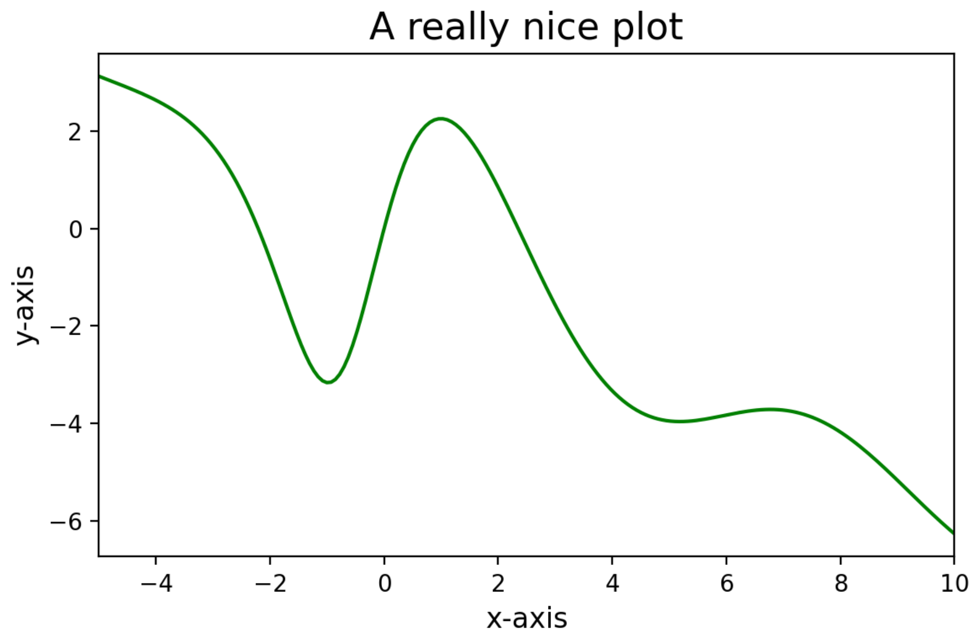
$y = \dfrac{5\sin x}{x + e^{-0.75x}} - \dfrac{3x}{5}$ over the range $-5 \le x \le 10$ with 200 values

- Green plot line that is 1.5 wide
- The title "A really nice plot""" with a fontsize of 16
- The x-axis label "x-axis" with a fontsize of 12
- The y-axis label "y-axis" with a fontsize of 12
- Plot limits of -5 and 10 for the x-axis

Plot with line color, style, and width settings

```
x = np.linspace(-5, 10, 200)
y = 5*np.sin(x)/(x + np.exp(-0.75*x)) - 3*x/5

fig, ax = plt.subplots(figsize=(6, 4), dpi=200)
ax.plot(x, y, '-g', lw=1.5)
ax.set_title('A really nice plot', fontsize=16)
ax.set_xlabel('x-axis', fontsize=12)
ax.set_ylabel('y-axis', fontsize=12)
ax.set_xlim(-5, 10)
plt.show()
```



7 Multiple Plots

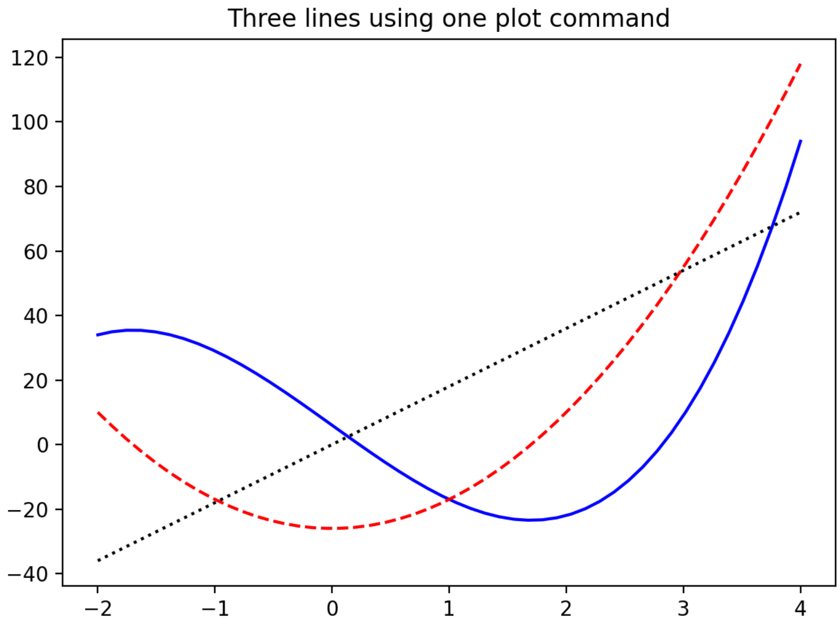
7.1 Multiple Plots on the Same Axes

To have more than one set of data in a single plot add more list pairs in the `.plot()` function. Color, line, and marker settings for each plot line should be placed after its pair of sequences. Another option is to create each plot separately using multiple `ax.plot()` commands; one for each set of data points then show them all. The later option is more flexible regarding formatting the lines. See the two following code blocks and plots for examples.

Three plots on in one axes using a single `.plot()` command

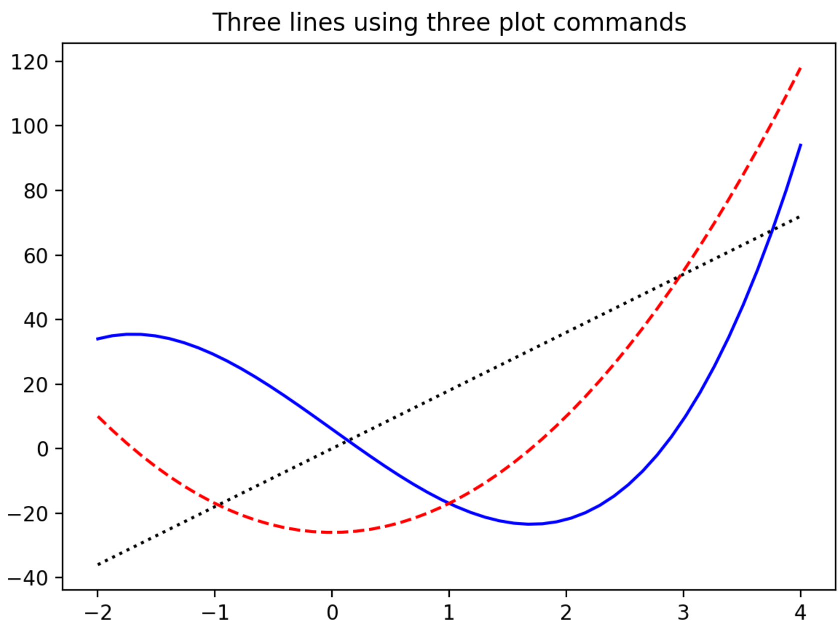
```
x = np.linspace(-2, 4)
y = 3*x**3 - 26*x + 6
yd = 9*x**2 - 26
ydd = 18*x

fig, ax = plt.subplots(figsize=(6,4.5), dpi=200)
ax.plot(x, y, '-b', x, yd, '--r', x, ydd, ':k')
ax.set_title('Three lines using one plot command')
plt.show()
```



Three plots in one axes using a separate .plot() command for each line

```
fig, ax = plt.subplots(figsize=(6,4.5), dpi=200)
ax.plot(x, y, '-b')
ax.plot(x, yd, '--r')
ax.plot(x, ydd, ':k')
ax.set_title('Three lines using three plot commands')
plt.show()
```



7.2 Multiple Plots on Separate Axes – Subplots

Multiple plots on separate axes are referred to as subplots.

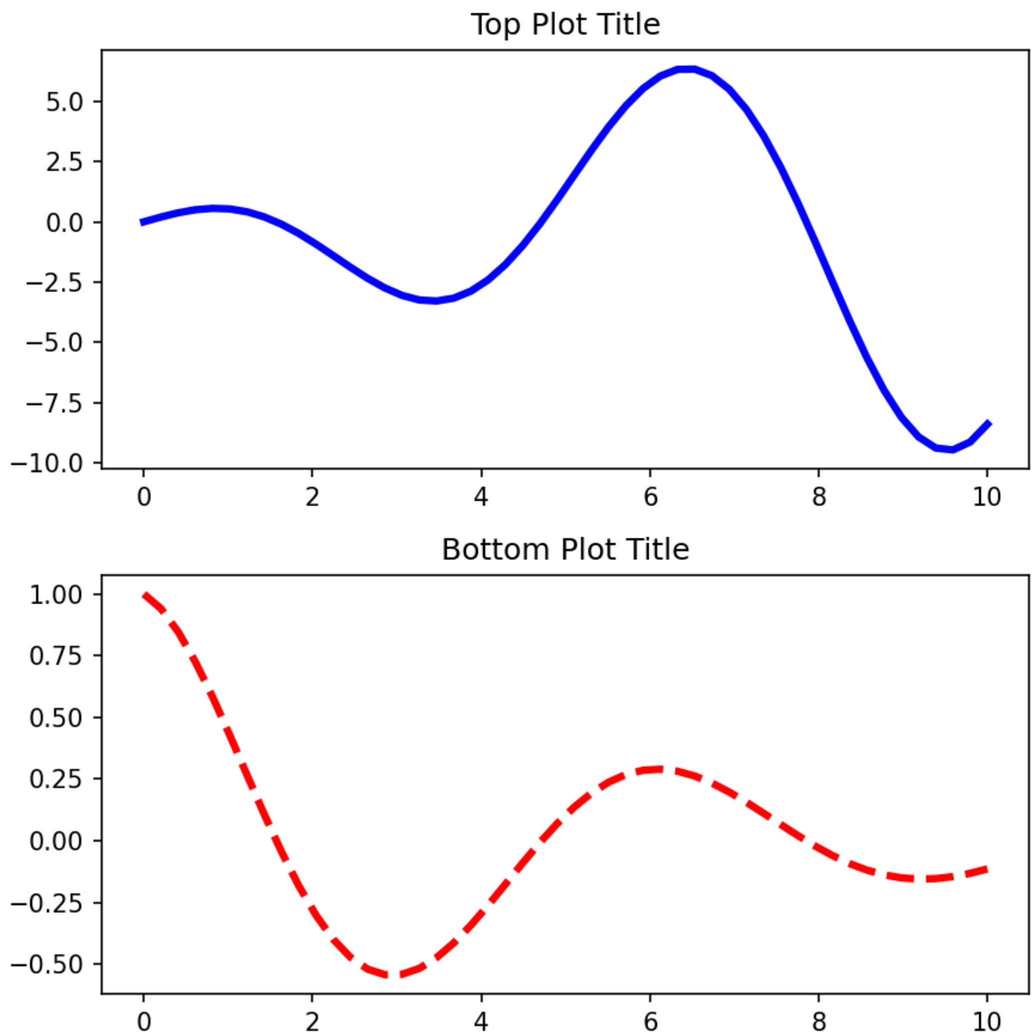
- The `plt.subplots()` method can be used to initialize a subplot with any number of rows or columns
  - Add the number of rows and columns desired inside the parentheses and separated by a comma
  - `plt.subplots(3, 1)` will be 3 plots high by 1 plot wide (3 rows by 1 column)
  - Can also use the keyword arguments `nrows=3` and `ncols=1` instead
- If using `fig.add_subplot()` to create axes objects then the argument used needs to be a 3-digit value representing...
  - Number of rows
  - Number of columns
  - Which subplot is the active one
  - `fig.add_subplot(211)` yields a subplot object 2 rows high by 1 column wide with the first subplot active
- If more than one axes object is created, then a list of axes objects is returned

Creating a pair of stacked subplots

```
x = np.linspace(0, 10)
y1 = x*np.cos(x)
y2 = np.exp(-0.2*x)*np.cos(x)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 6), dpi=150)
ax1.plot(x, y1, 'b-', lw=3)
ax2.plot(x, y2, 'r--', lw=3)
ax1.set_title('Top Plot Title')
ax2.set_title('Bottom Plot Title')
plt.tight_layout()
plt.show()
```

The line `fig, (ax1, ax2) = plt.subplots(2, 1, ...)` will assign the two subplot objects to the names `ax1` and `ax2` instead of to a single name for both. This way, plotting to the top (first) subplot uses `ax1.plot()` and plotting to the bottom (second) subplot uses `ax2.plot()`. An alternative would be to use a single name, such as `ax`, for pair of subplots. This method requires indexing to plot to each subplot; i.e. `ax[0].plot()` to plot to the top subplot. It is also a good idea to use `plt.tight_layout()` just before `plt.show()` to automatically clean up the spacing between and around the subplots.



8 A Polar Plot and a Histogram

Matplotlib supports the construction of much more than just x, y plots. Two of the many other plot types are histograms and ploar plots.

8.1 Polar plot example

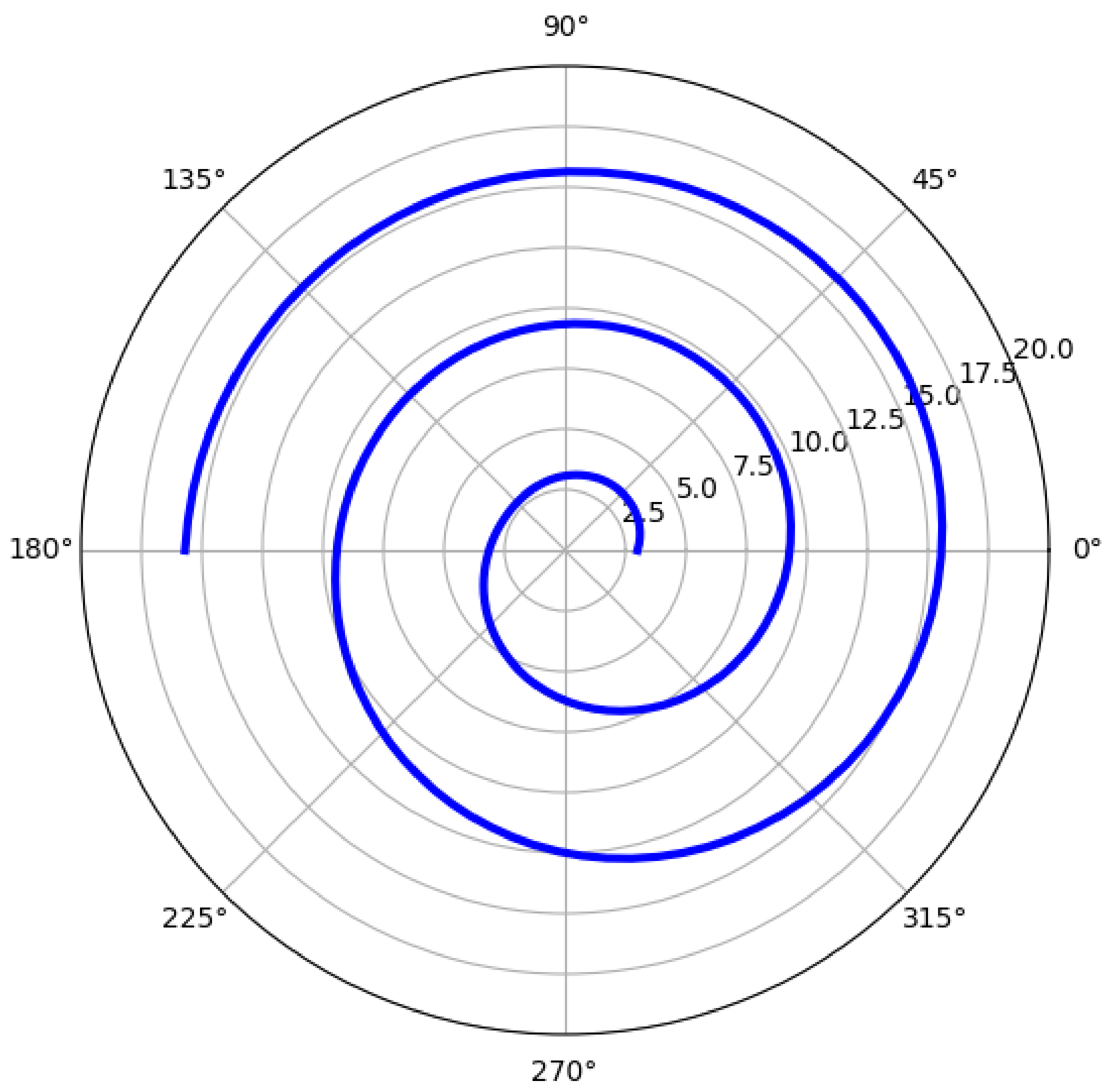
The following code block will create a polar plot. Notice that `ax.plot()` is still used in conjunction with `plt.figure()` and `fig.add_axes()`.

- Add the `polar=True` keyword argument to `fig.add_axes()` after the list with the size information
- The two arguments in `ax.plot()` are...
  - First: angle (in radians)
  - Second: radial value

A polar plot using `fig.add_axes()` with `polar=True`

```
t = np.linspace(0, 5*math.pi, 200) # 200 points from 0 to 5pi radians
r = 3*np.cos(0.5*t)**2 + t

fig = plt.figure(figsize=(6, 6), dpi=100)
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)
ax.plot(t, r, 'b', lw=3)
ax.set_ylim(0, 20) # ylim sets the radial limits for polar plots
plt.show()
```

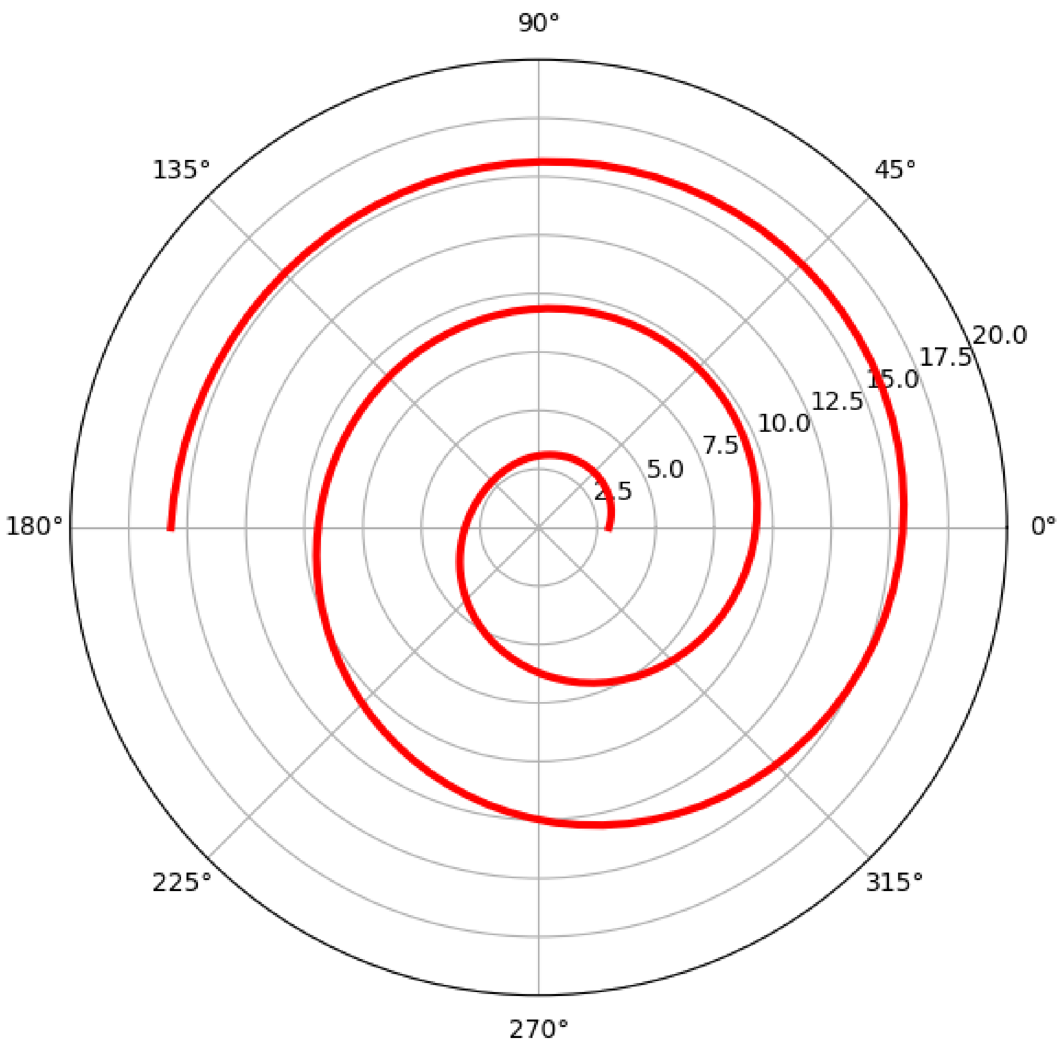


8.2 Another polar plot example

Another option for polar plots is to use `plt.subplots()` to create the figure and add the keyword argument `subplot_kw={'polar':True}`.

A polar plot using `plt.subplots()` with `subplot_kw={xṕolar:True’}`

```
fig, ax = plt.subplots(figsize=(6, 6), dpi=100,
                       subplot_kw={'polar':True})
ax.plot(t, r, 'r', lw=3)
ax.set_ylim(0, 20) # ylim sets the radial limits for polar plots
plt.tight_layout()
plt.show()
```

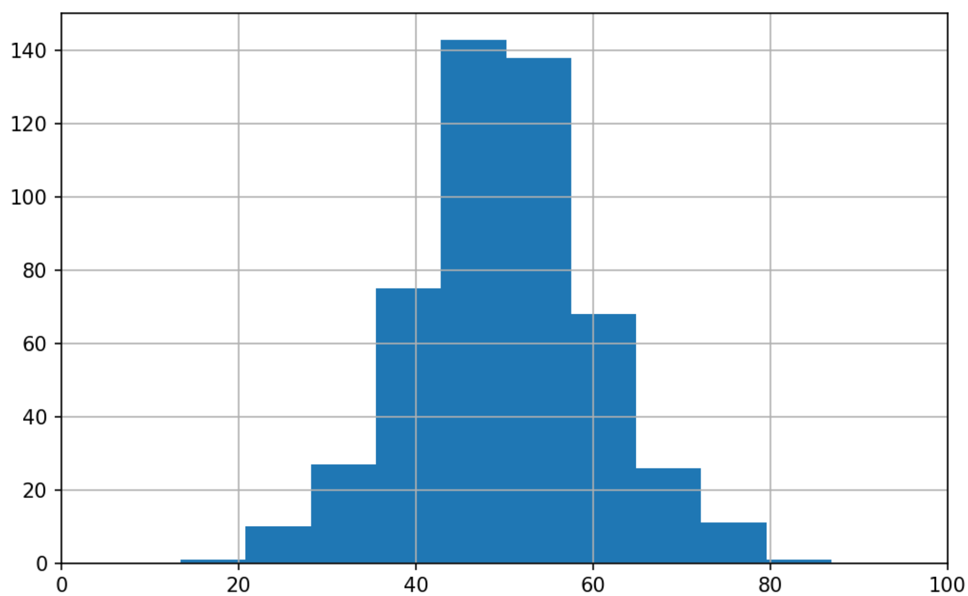


8.3 A histogram

Histograms are quite easy to make as well. The `np.random.normal()` function is used to create a list of random numbers that fit into a normal distribution. The `ax.hist()` function simply needs a list of values and the number of bins desired. Read about the command to see other options.

Histogram

```
mean = 50
std = 10
y = np.random.normal(mean, std, 500)
fig, ax = plt.subplots(figsize=(8, 5), dpi=150)
ax.hist(y, 10)        # 10 bins (buckets)
ax.set_xlim(0, 100)
ax.grid(True)
plt.show()
```



9 More Examples

9.1 Vibration

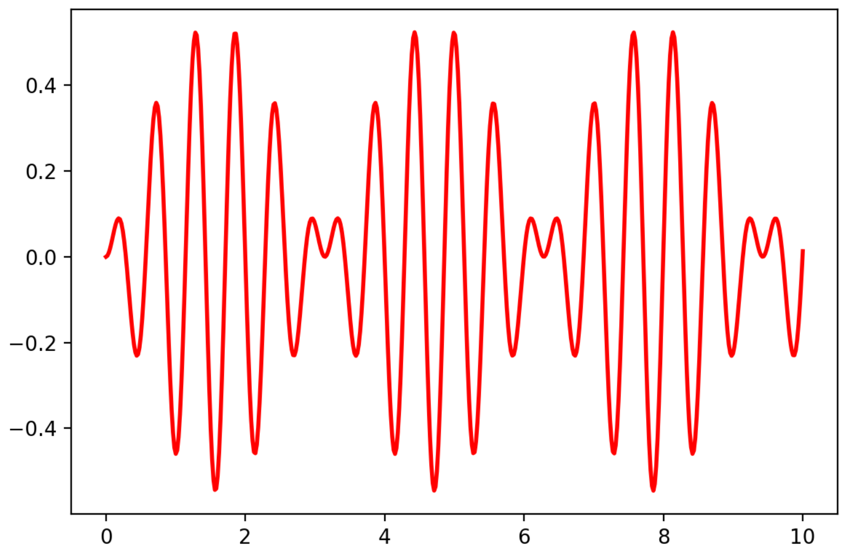
The following code block plots the following expression...

$$x(t) = \frac{2f_0}{\omega_n^2 - \omega^2}\ \sin\left((\omega_n - \omega)\,\frac{t}{2}\right)\sin\left((\omega_n + \omega)\,\frac{t}{2}\right)$$

for $0 \leq t \leq 10\text{s}$ with 500 values of t where $f_0 = 12\text{N/kg}$, $\omega_n = 10\text{rad/s}$, and $\omega = 12\text{rad/s}$.

Vibration plot

```
t = np.linspace(0, 10, 500)
f_0 = 12
w_n = 10
w = 12
x = 2*f_0/(w_n**2 - w**2)*np.sin((w_n - w)*t/2)*np.sin((w_n + w)*t/2)
fig, ax = plt.subplots(figsize=(6,4), dpi=200)
ax.plot(t, x, 'r', lw=2)
plt.show()
```



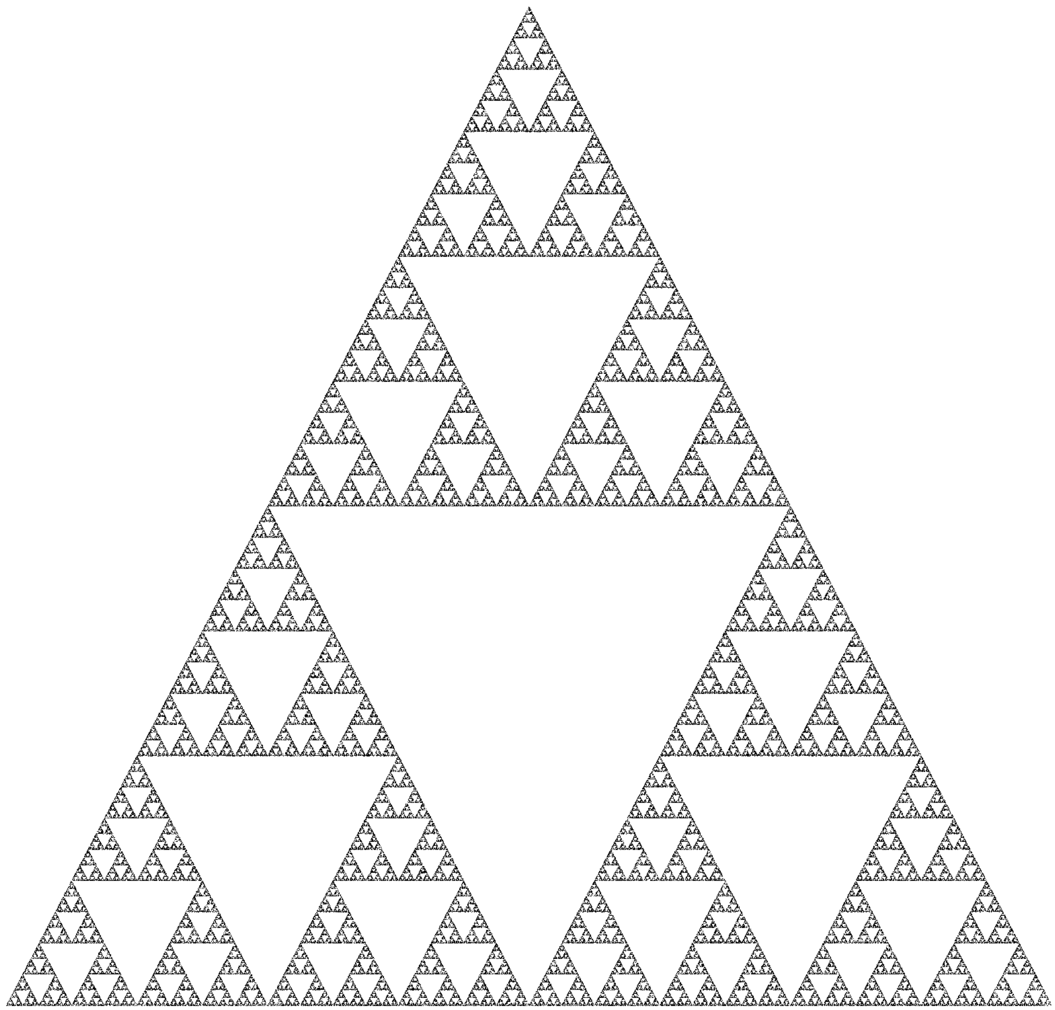
9.2 A Cool Math Plot

The following code cell uses `if-elif-else` statements and random values in a `for` loop to create the values for plotting **Sierpinski's Triangle** `https://en.wikipedia.org/wiki/Sierpinski_triangle`

Sierpinski's Triangle

```
from random import randint
x = [0]
y = [0]
n = 200000     # number of values to use
for k in range(1, n):
    choice = randint(0, 2)
    if choice == 0:
        x.append(0.5*x[-1])
        y.append(0.5*y[-1])
    elif choice == 1:
        x.append(0.5*x[-1] + 0.25)
        y.append(0.5*y[-1] + math.sqrt(3)/4)
    else:
        x.append(0.5*x[-1] + 0.5)
        y.append(0.5*y[-1])
fig, ax = plt.subplots(figsize=(6,6), dpi=300)
ax.plot(x, y, 'k,')
ax.set_title("Sierpinski's Triangle", fontsize=16)
ax.axis('off')
plt.show()
```

Sierpinski's Triangle



9.3 Na, Na, Na, Na, Na, Na, Na, Na, Na, Na, Na, Na, Na, . . .

. . . Batman

```
fig, ax = plt.subplots(figsize=(8,6), dpi=120)

x1 = np.linspace(-8, 8, 200)
y1 = 4*np.sqrt(-(x1/8)**2 + 1)
y2 = -4*np.sqrt(-(x1/8)**2 + 1)
ax.fill_between(x1, y1, y2, color='gold')
ax.plot(x1, y1, c='black', lw=3)
ax.plot(x1, y2, c='black', lw=3)

x3 = np.linspace(-7, -3, 100)
y3= 3*np.sqrt(-(x3/7)**2 + 1)
x4 = np.linspace(3, 7, 100)
y4 = 3*np.sqrt(-(x4/7)**2 + 1)
ax.fill_between(x3, y3, y2=0, color='black')
ax.fill_between(x4, y4, y2=0, color='black')

x5 = np.linspace(-7, -4, 100)
y5 = -3*np.sqrt(-(x5/7)**2 + 1)
x6 = np.linspace(4, 7, 100)
y6 = -3*np.sqrt(-(x6/7)**2 + 1)
ax.fill_between(x5, y5, y2=0, color='black')
ax.fill_between(x6, y6, y2=0, color='black')

x7 = np.linspace(-4, 4, 200)
y7 = (np.abs(x7/2) - (3*np.sqrt(33) - 7)/112*x7**2 +
      np.sqrt(1 - (np.abs(abs(x7) - 2) - 1)**2) - 3)
ax.fill_between(x7, y7, y2=0, color='black')

x8 = np.linspace(-1, -0.75, 10)
x9 = np.linspace(0.75, 1, 10)
y8 = 9 - 8*np.abs(x8)
y9 = 9 - 8*np.abs(x9)
ax.fill_between(x8, y8, y2=0, color='black')
ax.fill_between(x9, y9, y2=0, color='black')

x10 = np.linspace(-0.75, -0.5, 10)
x10 = np.append(x10, np.linspace(0.5, 0.75, 10))
y10 = 0.75 + 3*np.abs(x10)
ax.fill_between(x10, y10, y2=0, color='black')
```

```
x11 = np.linspace(-3, -1, 100)
y11 = 1.5 + 0.5*x11 - 6*np.sqrt(10)/14*(np.sqrt(3 - x11**2 - 2*x11) - 2)
x12 = np.linspace(1, 3, 100)
y12 = 1.5 - 0.5*x12 - 6*np.sqrt(10)/14*(np.sqrt(3 - x12**2 + 2*x12) - 2)
ax.fill_between(x11, y11, y2=0, color='black')
ax.fill_between(x12, y12, y2=0, color='black')

ax.axis([-9, 9, -5, 5])
ax.tick_params(axis='both', which='both', bottom=False,
               left=False, labelbottom=False, labelleft=False)
ax.set_facecolor('black')
plt.show()
```